# A Python Experiment Suite

**Thomas Rückstieß**
*Technische Universität München, Germany*
ruecksti@in.tum.de

**Jürgen Schmidhuber**
*Dalle Molle Institute for Artificial Intelligence, Lugano, Switzerland*
juergen@idsia.ch

### Abstract

We introduce the Python Experiment Suite, an open source software tool written in Python, that supports scientists, engineers and others to conduct automated generic software experiments on a larger scale with numerous features: parameter ranges and combinations can be evaluated automatically, where different experiment architectures (e.g. grid search) are available. The suite also takes care of logging results into files, can handle experiment interruption and continuation, for instance after process termination by power failure, supports execution on multiple cores and contains a convenient Python interface to retrieve the stored results. Configuration files ease the setup of complex experiments without modifying code, and various run-time options allow for a variety of use cases.

### Keywords

Scientific Software, Python, Experiments, Automated Evaluation, Grid Search

## 1. Introduction

Scientists, engineers, students and many others often find themselves running experiments or evaluations for their research and work. While the applications differ from field to field, the common factors are usually the same: run numerous experiments, repeat the same experiments many times, sometimes with slight variations, try a range of parameters, log and retrieve the results. A lot of time is usually spent on implementing and refining the experiments, but the actual experiment setup is often neglected, which bears the danger of introducing errors to the results and overwriting or losing results that have been very time-consuming to get in the first place. And implementing a proper experiment setup costs time, which, in the scientists' eyes, is better spent elsewhere. Only rarely do researchers publish their experiment framework as part of their scientific contributions, as e.g. (Beazley and Lomdahl, 1997).

Coming from the area of Machine Learning (see (Bishop, 2006) for an overview), our lab has been confronted with this problem in the past. As we develop new algorithms, a large part of our work consists of running comparisons and evaluations on new and existing methods, often with varying sets of parameters and many repetitions to average over

probabilistic outcomes. We therefore developed a tool that takes care of the whole experiment setup, while being transparent and easy to use and extend. We chose Python as the programming language, because it offers great packages, particularly suited for scientific computing (Oliphant, 2007): numpy, scipy and matplotlib are some of them (see also (Langtangen, 2008), in particular Chapter 4, on Python for numeric experiments and (Ousterhout, 1998) for an overview on general scripting languages for scientific computation).

The Python Experiment Suite, as we call our tool, is completely flexible in what experiments it should run. It is a wrapper around the experiments you want to conduct, and takes care of result logging, multi-core execution, parameter range evaluations and more. It has been developed over the last several years and has now matured to a state, where it is useable and useful to others as well. We have not seen any other tool specifically designed for this task, and hope that this software will save you time that you could spend on your work and research instead.

In the following sections, we will give a detailed description of the software, explain how to acquire and use the Experiment Suite, how to set up experiments, make use of the configuration files and finally demonstrate the API to retrieve the results.

## 2. Implementing a Python Experiment Suite

The Python Experiment Suite is developed in Python and requires Python 2.6 or higher, because it makes use of the multiprocessing module, which was only added in version 2.6. Another dependency is numpy (Oliphant, 2007). Other than that, the suite is a stand-alone package. For instructions on how to download and install the Experiment Suite package, please refer to the documentation, available on the github website, where the project is hosted: `http://github.com/rueckstiess/expsuite/`.

We start by giving a short overview of the definition of experiments in Section 2.1, before listing a typical workflow for the Experiment Suite in Section 2.2. This is also a checklist for the following sections. We then describe the two main functions of the suite, `reset()` and `iterate()`, and how they are implemented. Furthermore, the option of resuming interrupted experiments will be explained in Section 2.5.

### 2.1. Overview

The experiments created by the Python Experiment Suite have two levels of execution: the repetition level and the iteration level. This distinction is made because it is often required to repeat the same experiment several times (repetition level), and experiments are often of iterative nature themselves (e.g. an optimization problem that optimizes parameters over a number of steps). Both repetitions and iterations are controlled by two parameters, defined in the configuration file. The parameter `repetitions` specifies, how often the same experiment should be repeated. For deterministic experiments, where the outcome is always the same for one parameter set, its value is usually set to 1. The `iterations` parameter defines, for how many steps one experiment runs internally. If the experiment cannot be separated into consecutive steps but finishes in one single iteration, the value should be set to 1.

Code listing 1 demonstrates in a simplified manner the interaction between the two levels. Note that this code snippet is not from the actual Experiment Suite source code but only illustrates how the repetition and iteration levels interact. The repetition level corresponds to the outer loop of program execution. In the `reset()` method the steps to set up the experiment are executed (e.g. loading the parameters and data). Once set up, an experiment can then run for a certain number of iteration steps in the inner "iteration level", where the `iterate()` method is executed repeatedly (this is where the actual calculations happen).

Listing 1: Simplified execution loop of an experiment in the suite.

```
for r in range(repetitions):
      reset()
      for i in range(iterations):
            iterate()
```

Thus, the main implementation work that needs to be done to use the Experiment Suite goes in the two functions `reset()` and `iterate()`. What code exactly goes into these methods depends entirely on the experiment you want to set up. Section 2.4 goes into more detail on this question and also contains a simple example.

Experiments can be parameterized with arbitrary parameters, which are read from a configuration file. This gives you full flexibility over the experiments without the need to change the code every time. You can define multiple experiments in a file, which will be executed consecutively. In the config file you also specify how often the experiments are repeated and where the results are stored. Section 3 explains how the configuration files work and which options are available.

## 2.2. Typical Workflow

This list will give you a short overview of the tasks you need to do if you want to use the Experiment Suite. They will be explained in detail in the following sections and references back to this list are made throughout the paper.

(1) create a class which derives from the `PyExperimentSuite` class

(2) add the object creation and call to the `start()` method at the bottom of the script

(3) implement the `reset()` and `iterate()` methods

(4) (optional step) implement the `save_state()` and `restore_state()` methods and set the `restore_supported` flag to `True`

(5) create or edit the `experiments.cfg` file and add all the experiments and their parameters

(6) run the suite from the command line

(7) (optional step) after completion, open your Python console or create a Python script and use the built-in interface to access your results and visualize or post-process them

These phases are explained in the following sections. Most of them are required, with the exception of step (4) and (7). Step (4) is only necessary if interrupted experiments (e.g. by power failure or process termination) need to be resumed exactly where they were aborted, on iteration level. Section 2.5 contains more information about this step. Step number (7)

refers to the included Python API to access the results. Some useful methods are provided to ease access to interesting aspects of your data. Of course, the log files can be accessed and parsed without the API as well. More on the included API methods can be found in Section 4.

### 2.3. Getting Started

To start using the Python Experiment Suite for your experiments, it is recommended (but not essential), that a new folder is created for each new experiment setup. Create a new file in the folder and give it a suitable name, for example `suite.py`. Import the `PyExperimentSuite` class and create a new class that inherits from `PyExperimentSuite`. Next, add the empty method declarations for the two functions `reset()` and `iterate()` as shown in Listing 2. While `reset()` does not need a return value, `iterate()` needs to return a dictionary. For now, this can be the empty dictionary `{}`. We will fill out these empty methods further below. Finally, add the three lines at the bottom of the script for creating the `MySuite` object and starting it. If you're done with this task, you have completed points 1 and 2 from the workflow list in Section 2.2.

Listing 2: MySuite class definition with empty methods for later implementation.

```
from expsuite import PyExperimentSuite

class MySuite(PyExperimentSuite):

    def reset(self, params, rep):
        pass

    def iterate(self, params, rep, n):
        ret = {}
        return ret

if __name__ == '__main__':
    mysuite = MySuite()
    mysuite.start()
```

Running this script calls the object method `start()`, which parses the main configuration file (which will be covered below) and executes the defined experiments. If the executing computer has multiple cores available and if the run-time option `-n` (or `--numcores`) is not set to 1, multiple processes are spawned via the Python multiprocessing package to execute experiments in parallel. Otherwise, the experiments are executed one by one.

In this case, however, an error message will appear on execution of the `suite.py` script to notify you that the config file `experiments.cfg` could not be found. To remedy this, create a file called `experiments.cfg` in the same folder and add a default section with the parameters shown in Listing 3. These three parameters are necessary in all your configuration files and the Experiment Suite will complain if any of them are missing.

Listing 3: Configuration file experiments.cfg with default section.

```
[DEFAULT]
repetitions = 1
iterations = 100
path = 'results'
```

After saving the configuration file, the script can be called with `python suite.py` without any error messages. Since no experiments have been defined yet, the suite will just return to the command line.

## 2.4. Implementing reset() and iterate()

The main implementation work necessary to use the Experiment Suite goes in two methods: `reset()` and `iterate()`.

Let's start with the `reset()` method, which is called once for each experiment repetition. Here, all the required objects need to be created and initialized and necessary data needs to be loaded. Everything required during the experiment should be assigned to object variables (accessed by the object reference `self`) because the `iterate()` method doesn't have access to local variables.

As a running example throughout this paper, we want to verify a simple stochastic law, namely that the sample mean of random numbers drawn from a normal distribution does converge to the actual mean of the distribution. We use the `random` module of the numpy package and store the drawn numbers in a numpy array.

The `reset()` method needs to contain the code to initialize the array with the correct size and to seed the random number generator. Listing 4 shows how this is implemented in the `reset()` function. Note that the listing only shows a code snippet for the `reset()` definition. The full code example can be found under Listing 8.

Listing 4: Implementation of the reset() method for the Random Numbers example

```
def reset(self, params, rep):
    # initialize array
    self.numbers = zeros(params['iterations'])

    # seed random number generator
    random.seed(params['seed'])
```

The array is initialized with a size of the number of iterations and assigned to an object variable `self.numbers` because we will need to access it later on in the `iterate()` method.

All parameters defined in the configuration file can be accessed through the dictionary `params`, which is passed into all the relevant method of the `PyExperimentSuite` class. In this example we access the number of iterations from the `params` dictionary and also make the random seed a parameter of the experiment, which needs to be defined in the configuration file `experiments.cfg`. Listing 6 shows the updated configuration file, already including two experiments called *normalized* and *highstd*.

The additional parameter `rep` that is passed to the `reset()` method contains the current repetition number, between 0 and `params['repetitions']-1`.

Next we will look at the `iterate()` method. It will be called repeatedly during the execution of one single repetition of one experiment. It can be seen as a single step or cycle within the experiment. Most experiments have a natural separation into repeated iterations already. If your experiment does not seem to be iterative, you might have to introduce some separation into steps manually. If you feel like this is not possible at all, you can also use a single iteration for the whole experiment, setting the `iterations` parameter in the config file to 1.

Just as in the `reset()` method, the `params` dictionary is also passed to the `iterate()` method. The second parameter passed to this method is `rep`, which indicates the current repetition, starting from 0. `iterate()` has a third parameter, `n`, which represents the current iteration index. The variable `n` starts from 0 and counts up to `params['iterations']-1`.

While the `reset()` method did not return any values, the `iterate()` method is expected to return a dictionary. The dictionary should contain the relevant information that you want to store in the log files. This can be some measure of progress of your experiment, an error, some identifying strings or the running index `n`.

To continue the Random Numbers example, we will now execute one step in the `iterate()` function. We draw a normally distributed random number, parameterized by the two config file parameters `mean` and `std`, then calculate the sample mean of all numbers drawn this far and calculate the offset to the real mean. We want to log some results, namely the current iteration number `n`, the drawn random number, the sample mean and the current offset. We therefore add all these values to the return dictionary. The keys in the dictionary are strings, with which we will later be able to retrieve the results. Use unique, descriptive labels for these keys.

Listing 5 demonstrates how to achieve this. Again, this is only the code snippet for the `iterate()` method. The full code example can be found under Listing 8.

Listing 5: Implementation of the iterate() method for the Random Numbers example

```
def iterate(self, params, rep, n):
    # draw normally distributed random number
    self.numbers[n] = random.normal(params['mean'], params['std'])

    # calculate sample mean and offset
    samplemean = mean(self.numbers[:n])
    offset = abs(params['mean']-samplemean)

    # return dictionary
    ret = {'n':n, 'number':self.numbers[n],
        'samplemean':samplemean, 'offset':offset}

    return ret
```

Besides the three mandatory parameters `repetitions`, `iterations` and `path`, we used three more parameters in the code example in Listings 4 and 5: `seed` for the random number generator seed, and `mean` and `std` to parameterize the normal distribution. These parameters have to be defined in the config file. We will make seed a default parameter that we leave

unchanged for now, and define two experiments, called *normalized* and *highstd* each with their own section in square brackets and with different values for the `std` parameter. We also increased the number of iterations to 10000 for this example. Listing 6 shows the final configuration file for this example.

Listing 6: Configuration file for the random numbers example with 2 experiments

```
[DEFAULT]
repetitions = 1
iterations = 10000
path = 'results'

seed = None

[normalized]
mean = 0.0
std = 1.0

[highstd]
mean = 0.0
std = 5.0
```

By implementing the `reset()` and `iterate()` methods, you have completed point 3 from the workflow list in Section 2.2. We also created the configuration file `experiments.cfg`, which completes point 5 from the workflow list.

## 2.5. Optional Resume Functionality

If you interrupt the suite while it is executing the experiments, by killing the process or even by a power failure, the suite is left in a somewhat undefined state.

Without any further implementation work, all it can do is to abandon the already executed iterations in the current repetition and start the repetition again. The Suite does that automatically: it will delete the already logged iterations of the current repetition (or repetitions, if you use multiple cores) and restart them from iteration step 0, by calling the `reset()` method again. Already completed repetitions are of course not affected and will remain in the log files.

Still, some experiments can be very time-consuming and even losing a few iteration steps might be inacceptable. In this case, you need to set the class variable `restore_supported` to `True` and implement a few more lines of code, namely the two methods `save_state()` and `restore_state()`. The `save_state()` method is called after each iteration step has completed, and its task is to save all relevant information needed to continue from this iteration step. This can be done by marshalling[1] crucial objects to a persistant memory store (e.g. a file or database), or otherwise saving the current state of the experiment.

[1] also called serializing in the Java community and pickling in the C and Python community. `pickle`, and its faster C implementation `cPickle` are Python packages that can serialize objects and save them to disk, and restore (unpickle) the files and create objects again. A full documentation of the pickle and cPickle packages can be found at

If the variable `restore_supported` is set to `True`, the suite checks upon start if there are unfinished experiments on iteration level. It then calls the `restore_state()` method with the appropriate parameters, repetition and iteration index. The task of the `restore_state()` function is to load the correct data from files and restore the class objects to that state.

Listing 7 shows, how these functions are implemented for the Random Numbers example. Basically, all the information required to resume the experiment where left off is the list of numbers drawn in previous iterations. In out example, the `save_state()` method thus needs to save the `self.numbers` array, and the `restore_state()` method needs to load it again.

If the script is executed now, and aborted or killed during runtime, the most data you could lose is one single iteration (the one that was currently being calculated, before the results could have been saved to disk). Listing 8 provides the full code for the Random Numbers example.

Listing 7: Save and restore methods for the Random Number example.

```
restore_supported = True

def save_state(self, params, rep, n):
    # save array as binary file
    save(os.path.join(params['path'], params['name'],
        'array_%i.npy'%rep), self.numbers)

def restore_state(self, params, rep, n):
    # load array from file
    self.numbers = load(os.path.join(params['path'],
        params['name'], 'array_%i.npy'%rep))
```

Listing 8: Complete code listing for Random Numbers example.

```
from expsuite import PyExperimentSuite
from numpy import *
import os

class MySuite(PyExperimentSuite):

    restore_supported = True

    def reset(self, params, rep):
        # initialize array
        self.numbers = zeros(params['iterations'])

        # seed random number generator
        random.seed(params['seed'])


    def iterate(self, params, rep, n):
        # draw normally distributed random number
        self.numbers[n] = random.normal(params['mean'],
            params['std'])

        # calculate sample mean and offset
```

`http://docs.python.org/library/pickle.html`. Other ways of saving data may be used as well, e.g. XML exports.

```
        samplemean = mean(self.numbers[:n])
        offset = abs(params['mean']-samplemean)

        # return dictionary
        ret = {'n':n, 'number':self.numbers[n],
            'samplemean':samplemean, 'offset':offset}

        return ret

    def save_state(self, params, rep, n):
        # save array as binary file
        save(os.path.join(params['path'], params['name'],
            'array_%i.npy'%rep), self.numbers)

    def restore_state(self, params, rep, n):
        # load array from file
        self.numbers = load(os.path.join(params['path'],
            params['name'], 'array_%i.npy'%rep))


if __name__ == '__main__':
    mysuite = MySuite()
    mysuite.start()
```

## 3. Parameter Configuration and Run-Time Options

This Section will give more details about the configuration file, which is workflow step 5 from Section 2.2, and available options on run-time for workflow step 6.

### 3.1. The Configuration File

The configuration file (`experiments.cfg` in the same folder by default) is separated in different sections that start with a section header in square brackets, like `[sectionname]`. Following the section header are entries of the form `name=value`, each on a single line. Leading whitespace is removed from values. Lines beginning with `#` or `;` are ignored and can be used as comments. The configuration script is parsed by the Python module ConfigParser[2] and follows its syntax.

A special section with header `[DEFAULT]` can be provided. Any other section will inherit all key-value pairs from this section, unless it defines another value for an existing key, in which case the new value will be chosen instead.

Acceptable values include any type that can be evaluated to a Python int, float, string, or object. In fact, `PyExperimentSuite` will try to evaluate the given value using the Python function `eval()`, and if no errors occur, interpret it as its Python pendant. This means that everything that can be evaluated to an integer will be treated as one, the same is true for floats. More complex values, like `sin(2.)` or `0.5*pi` are also valid expressions.

Everything that cannot be evaluated without raising an exception will be interpreted as a string. This means that string values with and without quotation marks are possible. As an example, the two parameter definitions `name=peter` and `name="peter"` will both evaluate to a dictionary entry in the `params` dictionary with key `name` and a string value `peter`. To avoid

[2] `http://docs.python.org/library/configparser.html`

confusion with strings equal to Python function names, however, it is recommended to add quotation marks around strings.

Lists and iterable objects play a special role as explained in Section 3.2 below and will, by default, not be passed to the parameters as their Python equivalents.

Listing 9: Examples for valid key-value pairs for the configuration file

```
[DEFAULT]
path = ./results/
repetitions = 1
iterations = 100

seed = None

[experiment1]
mean = 0.1
std = [0.5, 1.0, 1.5]

[experiment2]
experiment = 'grid'
mean = arange(0.0, 1.0, 0.2)
std = [0.5, 1.0, 2.0]

[experiment3]
experiment = 'list'
mean = arange(0.0, 1.0, 0.2)
std = [0.5, 1.0, 2.0]

[experiment4]
experiment = 'single'
mean = arange(0.0, 1.0, 0.2)
std = [0.5, 1.0, 2.0]
```

### 3.2. Evaluating Parameter Ranges

If your value evaluates to a Python object that is iterable (e.g. lists, numpy arrays, generator objects, etc.) the Suite will not assign the object to the parameter by default. Instead, it assumes that you want to try a range of values in separate experiments. This is very handy if you need to find the optimal value for certain parameters, because you can just define the range and let the Experiment Suite do all the testing. Per default, the elements of iterable objects are passed to the parameter individually in separate sub-experiments, while all other non-iterable parameters are kept constant. If you need to assign a list or iterable object to a parameter and do not want the Suite to create several paramter range experiments in this way, check out the experiment type `'single'` in Section 3.3 below.

### 3.3. Parameter Range Combinations

For more than one parameter that evaluates to an iterable object, two choices are available: try every single combination of all parameter choices or run successive experiments with the parameters of the same list indices. The different experiment types can be set by the configuration parameter `experiment`. Figure 1 explains the difference graphically.

The first choice is called a *grid* experiment, and is also set as the default. The Experiment Suite will create sub-experiments for every single combination of all parameters.

Experiment 2 in Listing 9 defines two iterable parameters, `mean` and `std`. Numpy's `arange()` evaluates to a list with the values 0.0, 0.2, 0.4, 0.6 and 0.8 for `mean`, and `std` takes on the values of 0.5, 1.0 and 2.0. In total, the Suite will run 5 x 3 = 15 experiments. Note the additional parameter `experiment`, which is set to the value `'grid'`. In this case, the `experiment` parameter could have been omitted, since the grid experiment is the default.

The second option for running an experiment with parameter range combinations is a *list* experiment, and the `experiment` parameter is set to the value `'list'`. Here, each experiment will contain values from both lists at the same indices. The number of sub-experiments created by this process is the length of the shortest list. In case of experiment 3 in Listing 9, the Suite will create 3 experiments with the following parameters:

```
mean = 0.0, std = 0.5
mean = 0.2, std = 1.0
mean = 0.4, std = 2.0
```

The list for `mean` would contain more values, but since there are not corresponding values in the `std` list, the other experiments are discarded.

Experiments are not limited to two lists but can have arbitrary numbers of iterable objects. Just keep in mind that in the grid case, the number of experiments is the product of all list lengths, and in the list case, it is the length of the shortest list.

If you don't want to create multiple experiments but use the assigned iterable object as value, use `'single'` as the `experiment` value.

### 3.4. Run-Time Options

When calling your suite.py script from the command line, several options are available. In this section, we will describe some of them and explain their usage. For a full list of options, refer to the documentation. After reading this chapter, you are able to complete workflow step 6 of Section 2.2.

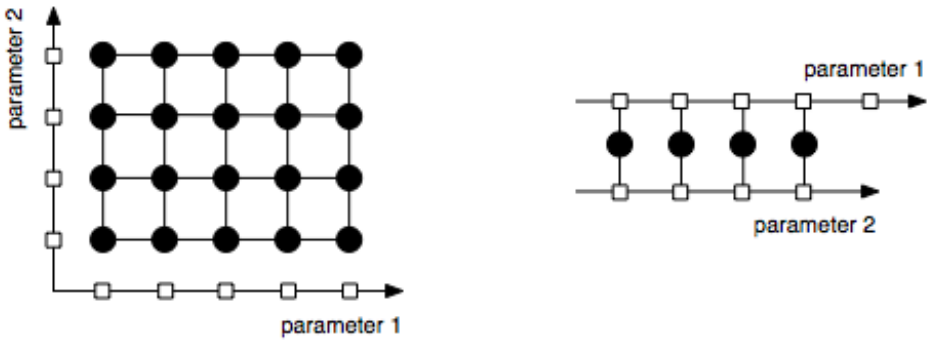


Figure 1: Two different types of experiments with parameter range combinations. The white squares represent discrete values of a parameter range, which is shown as an arrow. The black circles are experiments with a certain parameter combination. On the left side, a grid experiment is evaluated. On the right side, a list experiment is shown.

A useful run-time option is the `-h` or `--help` flag. Calling the script with that option will present a list of all the available command line options. Running the Python script with this flag will not execute any experiments but return to the prompt immediately.

The `-n <number>` or `--numcores=<number>` option lets you restrict PyExperimentSuite to a certain number of cores. If you specify this option, the Suite will at most use the number of cores given. If this option is not present, it will use all the available cores it can find on the local machine.

If you don't want to execute all experiments defined in the config file but only one or a few selected ones, you can use the `-e <experiment>` or `--experiment=<experiment>` option. Specify each experiment you want to run explicitly with a separate `-e` option. If this option is not present, all defined experiments will be executed. Note that this option is unrelated to the experiment type (grid, list, single) which is set in the configuration file.

The browse option `-b` or `--browse` does not execute the script (no experiments are started or continued) but returns a list of current experiments and additional information. The lower-case version of this option tells you each experiment's name, start and end time, the number of repetitions and iterations, and the progress of the experiment in percent. The upper-case version `-B` or `--Browse` is more verbose and additionally tells you all the parameters for each experiment. Similar to the browse option, the progress option `-p` shows a list of experiments and a progress bar of their completion.

## 4. Result Retrieval

This section explains what you can do after your experiments have finished. It will explain where the log files are stored, what you will find in the log files, and how you can easily retrieve any relevant information within Python, without writing your own log file parser.

### 4.1. Log Files

The log files that the Experiment Suite creates contain the key-value pairs that the `iterate()` function (ref. Section 2.4) returned as dictionary. Each line of a log file corresponds to one single iteration. They key-value pairs are stored in this format:

```
key1:value1 key2:value2 key3:value3 ...
```

For each repetition, a separate log file is created, and each log file is named `#.log` where `#` is the repetition number, starting with 0. The logfile is located depending on the defined `path` and `experiment`, for example at at `./results/experiment1/0.log`.

As the order of Python dictionary entries is unspecified, so is the order of the key-value pairs in the log file[3]. The log file of our ongoing Random Numbers example could look like this (note the `n` key at the end of the line rather than the beginning, as defined in the `iterate()` method):

```
samplemean:-1.3254 number:0.5941 offset:1.3254 n:1
samplemean:-0.3656 number:2.6571 offset:0.3656 n:2
samplemean:0.6419 number:-1.4690 offset:0.6419 n:3
samplemean:0.1141 number:0.4120 offset:0.1141 n:4
...
```

If your experiment contained any parameter range evaluations with lists or other iterable objects (see Section 3.2 and 3.3), PyExperimentSuite will have created sub-directories under `./path/name/` with unique names based on the evaluated parameter combination. Within the sub-directories, the log files are stored as described before.

The log files can be used to post-process your results, or plot or otherwise display them. The Experiment Suite ships with a build-in parser already, that is accessible via a Python interface. The next section explains, how to retrieve your stored results.

### 4.2. Python Interface

The Python Experiment Suite comes with several functions that can be used to retrieve data after the experiments have run. This section describes the final workflow step 7 from the list in Section 2.2.

The names of all functions intended to use for data retrieval start with the prefix `get_`. Some of them are described in this section, for a full list please refer to the Experiment Suite documentation.

The first retrieval function you might use is `get_exps()`. This function goes through all existing subdirectories and locates your experiments. It returns the full paths of all existing experiments (finished or not) under the current directory as a list. The function has an

[3] Python 2.7 introduced ordered dictionaries that remember the order of inserted keys. Future versions of the Python Experiment Suite might use these ordered dictionaries. To keep the required Python version as low as possible, we decided to use normal dictionaries here and stick with version 2.6. As the data retrieval API imports the config files back into Python dictionaries, this is not a big issue here.

optional parameter `path` that specifies the directory of where to start with the search. All other data retrieval functions, require an experiment identifier `exp` in their function call, which is the path and name of the experiment, e.g. `./results/experiment1`. The above function returns this path, so you can use it for the functions that follow below.

The most atomic data retrieval task is to access a single value of a single iteration.

```
>>> mysuite.get_value(exp, rep, tags, which)
```

You could, for example, want to find the value of `offset` after 25 iterations in the first repetition of experiment *experiment1*. The function `get_value()` does exactly that. It takes four parameters: `exp` is the directory of the experiment you would like to access, `rep` specifies the repetition number. `tags` can be a string or a list of strings. If it is a string, it needs to contain the key which you would like to access. This is the key you added to the dictionary in method `iterate()`. If you want to access several keys at once, pass a list of strings to the function, and if you want to retrieve all keys, simply pass the string `'all'`. Finally, you need to specify, `which` value(s) you would like to access. The choices are the strings `'last'`, `'min'`, `'max'` or an integer. `'last'` will return the value from the very last iteration. `'min'` and `'max'` return the minimal or maximal value over all iterations. If you pass an integer to the `which` argument, you will get the value at this specific iteration. For the afore-mentioned example, the call would be

```
>>> mysuite.get_value('./results/experiment1', 0, 'offset', 25)
```

If you requested only one key, the return value will be a scalar. If you requested several or all keys, you will get a dictionary with key-value pairs as a result.

Retrieving a single value from a sub-experiment created by parameter ranges works the same, except the experiment location is a bit longer. Use the full experiment path `./path/name/sub-exp` to retrieve single values.

There is, however, another function that is made to work with parameter ranges, and it is particular useful with grid experiments. If you have tested a grid of several parameters, and you would like to retrieve values along one particular parameter axis and fixing the other ranges, the function `get_values_fix_params()` might be of use to you.

```
>>> mysuite.get_values_fix_params(exp, rep, tags, which, **kwargs)
```

It uses the same syntax as `get_value()` but has an additional `**kwargs` argument, which means you can pass an arbitrary number of additional keyword-value pairs in the function. The keywords expected here are the keys you would like to keep fixed to a certain value. Basically, for each keyword you add, you slice the (possibly multi-dimensional) grid along one axis and only consider experiments that lie on that axis. Refer to the left side of Figure 1 again for visual support.

Let's have a look at experiment 2 from Listing 9, which defined two parameter ranges in a grid fashion:

```
mean = arange(0.0, 1.0, 0.2)
std = [0.5, 1.0, 2.0]
```

This definition will create 15 experiments (because `mean` evaluates to `[0.0, 0.2, 0.4, 0.6, 0.8]`). If this was part of our Random Numbers example, and we were looking for the

smallest offset under the condition that the mean was 0.2, this is how we would call the function:

```
>>> values, params = mysuite.get_values_fix_params(
        './results/experiment2', 0, 'offset', 'min', mean=0.2)
```

As you can see in the function call, this function actually returns a tuple of two different things: the actual values, and the experiment parameters. Both return values are lists of equal lengths. The first one contains results in the form that you would expect from `get_value()`: either scalars or dictionaries of key-value pairs. the second return value contains equally many dictionaries: each dictionary consists of all the parameters of the experiments that matched the criteria given by `**kwargs`.

You can add as few or many conditions to the function as you like, not just one. Each condition limits the returned experiments and values further. If you call the function without any conditions, it will go over all sub-experiments. As an example, if you evaluated several different parameter ranges in a grid and just want to find the overall lowest offset, call this function without any `**kwargs` arguments.

Another common task with the Experiment Suite is to retrieve a whole history over all iterations of a certain key, for example if you want to plot how the error of an optimization problem slowly decreased with each iteration. The function `get_history()` can help you with this.

```
>>> mysuite.get_history(exp, rep, tags)
```

This function expects the location of your experiment, the index of the repetition you would like to access, and the keys you want to retrieve, either a single string, a list of strings, or the string `'all'`. It then returns a list of all values of that key, or a dictionary of lists with the corresponding keys.

In case you want to plot the history with the separate module matplotlib, you could call:

```
>>> plt.plot(mysuite.get_history('./results/experiment1', 0,
        'offset'), '-o', label='mean offset')
```

Just as with the `get_value()` method described above, histories can also be returned from parameter range experiments. the function `get_histories_fix_params()` retrieves several histories over a set of parameter range experiments with additional constraints.

There is another useful history-retrieving function, which is most useful if you executed many repetitions of the same experiment. One reason to do this, is because your experiment might be of stochastic nature and you want to average over all these repetitions.

```
>>> mysuite.get_histories_over_repetitions(exp, tags, aggregate)
```

This function requires the location of your experiment, the key(s) you are interested in (again as single string, list of strings or the string `'all'`), and an aggregation function. Typical aggregation functions could be `sum`, `mean`, `max`, etc. The function will not return the history of one single repetition, but use all repetitions and map the values of each repetition to the aggregation function. Let's say the aggregation function is called `aggr()` and the value at repetition r and iteration i is defined as $v_i^r$. Repetitions range from 0 to R and iterations range from 0 to I. The result is then a new list that contains values with same iteration indices, to which `aggr()` has been applied:

$$[\text{aggr}(v_0^0, v_0^1, \ldots, v_0^R), \text{aggr}(v_1^0, v_1^1, \ldots, v_1^R), \ldots, \text{aggr}(v_I^0, v_I^1, \ldots, v_I^R)]$$

One common use of this function is to average histories over all repetitions with the aggregation function `mean` (imported from numpy):

```
>>> mysuite.get_histories_over_repetitions(
        './results/experiment1', 'offset', mean)
```

## 5. Limitations

One of the obvious drawbacks of the Python Experiment Suite is its limitation to one single computer. While it makes use of Python's multiprocessing package and can spread repetitions and experiments over multiple cores, there is currently no way to extend this to several machines. An extension for cluster computing using the Parallel Python (pp) module is planned and will be available in a future version.

Currently, it is not trivial to use the Python Experiment Suite for non-pythonic experiments. While it is possible to call shell scripts from within Python to start experiments, the feedback for log files is not obvious. System packages like Python's `os` and `sys` might be of help, or the `Popen3` and `Popen4` classes, which can access the standard output and error streams of the programs.

Another limitation is that experiment types like *grid*, *list* and *single* cannot be combined to run even more complex setups. This limitation was deliberately chosen to simplify the configuration process of experiments. A more flexible approach would necessarily come with a more complex experiment definition, which most users have no need for.

Finally, the current experiment types are static, but machine learning provides methods of flexible experiment design, like active learning (Cohn et al., 1994; Milano et al., 2001). In those experiments the result of one experiment defines the parameters of the next experiment. Support for dynamic experiment types are subject of ongoing development.

## 6. Conclusion

We introduced the Python Experiment Suite, a tool that facilitates experiment design and execution with many features. It offers an easy, convenient way to run scientific experiments in Python, while taking care of result logging, multi-core execution and parameter range evaluations. While the tool still has its limitations, we hope that a large group of people will find it useful and save time that they can spend in their work or research instead.

## References

Beazley, D. and Lomdahl, P. 1997. Building flexible large-scale scientific computing applications with scripting languages. In 8th SIAM Conference on Parallel Processing for Scientific Computing, 14–17.

Bishop, C. 2006. Pattern recognition and machine learning. Springer.

Cohn, D., Atlas, L., and Ladner, R. 1994. Improving generalization with active learning. Machine Learning 15, 2, 201–221.

Langtangen, H. 2008. Python scripting for computational science. Springer.

Milano, M., Schmidhuber, J., and Koumoutsakos, P. 2001. Active learning with adaptive grids. In Artificial Neural Networks (ICANN), 436–442.

Oliphant, T. 2007. Python for scientific computing. Computing in Science & Engineering, 10–20.

Ousterhout, J. 1998. Scripting: Higher-level programming for the 21st century. IEEE Computer 31, 3, 23–30.